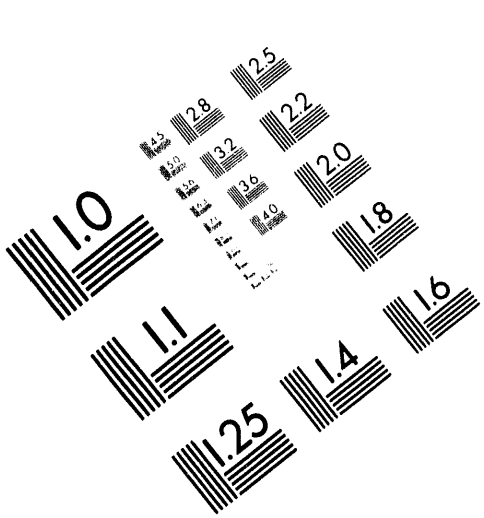
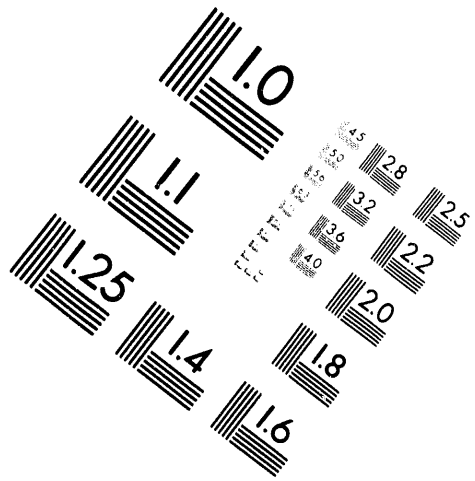
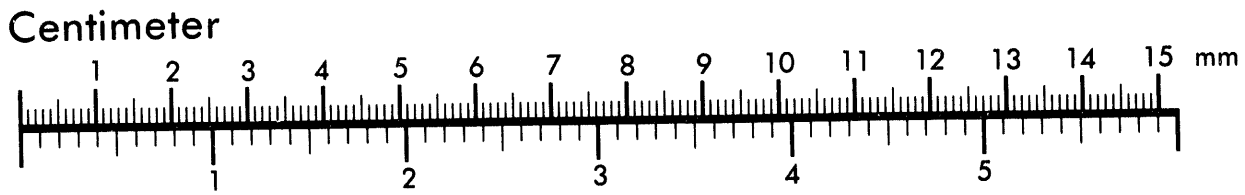


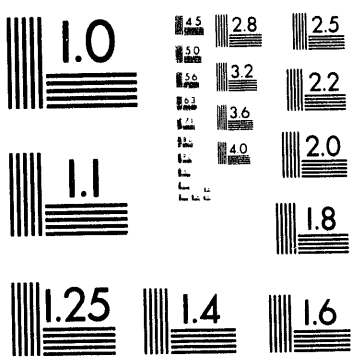
Association for Information and Image Management

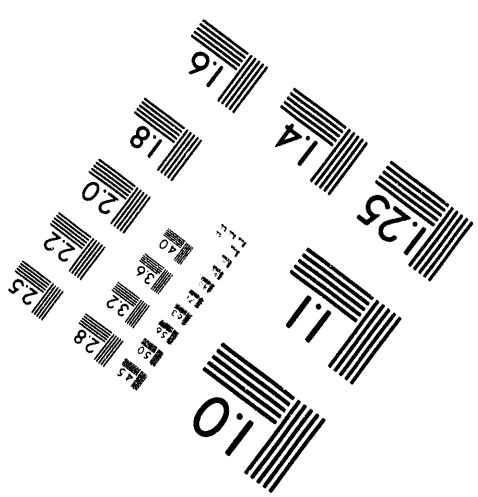
1100 Wayne Avenue, Suite 1100
Silver Spring, Maryland 20910

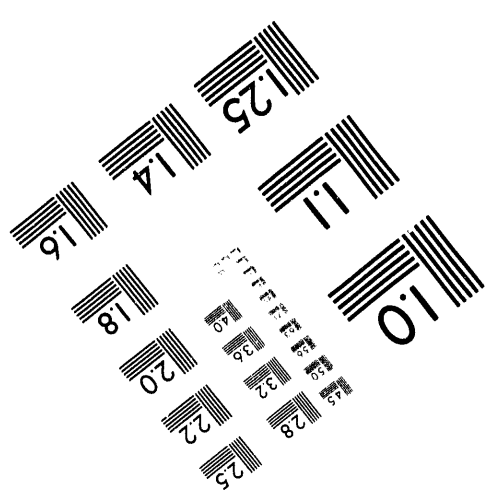
301/587-8202











1 of 1

UC-405

SAND94-8233
Unlimited Release
Printed July 1994

DAVE: A Plug and Play Model for Distributed Multimedia Application Development

Robert F. Mines, Jerrold A. Friesen, Christine L. Yang
Distributed System Research Department
Sandia National Laboratories/California

ABSTRACT

This paper presents a model being used for the development of distributed multimedia applications. The Distributed Audio Video Environment (DAVE) was designed to support the development of a wide range of distributed applications. The implementation of this model is described. DAVE is unique in that it combines a simple "plug and play" programming interface, supports both centralized and fully distributed applications, provides device and media extensibility, promotes object reuseability, and supports interoperability and network independence. This model enables application developers to easily develop distributed multimedia applications and create reusable multimedia toolkits. DAVE was designed for developing applications such as video conferencing, media archival, remote process control, and distance learning.

DAVE: A PLUG AND PLAY MODEL FOR DISTRIBUTED MULTIMEDIA APPLICATION DEVELOPMENT

1. INTRODUCTION

Significant advances have been made in distributed computing and in the development of multimedia ready computers and related hardware. Models for the integration of these two technologies are not yet available. Software applications, such as desktop video conferencing, that take advantage of this new hardware are being introduced. What is lacking are software models that integrate these new technologies into a distributed multimedia environment. Distributed Audio Video Environment (DAVE) is a model that provides for that integration. DAVE was designed to support the development of a wide range of distributed applications.

Our approach combines object-oriented analysis and design, distributed computing, and multimedia technologies to develop a heterogeneous, distributed multimedia development environment. DAVE was implemented using standard UNIX workstation components and traditional IP networking for all interhost communications. In the past, distributed environments (e.g., Rapport[1]) relied on means such as coax cable or telephony rather than IP for media transport. Our goal was to leverage off existing workstation and network infrastructures to increase the level of interoperability and availability.

The strengths of the DAVE model come from the programming interface and the ability to provide a high level of abstraction for devices through object oriented techniques. The programming paradigm allows application developers to treat media devices (e.g., cameras and microphones) as distributed resources akin to the way graphics and windows are used today on workstations. This flexibility and level of accessibility allows developers to easily integrate multimedia into our existing distributed environment. Through inheritance and data independence, developers are able to define additional devices and media types and integrate them into DAVE. We found when porting our video conferencing application to a Sun from an SGI required only the development of a relatively small amount of device specific code and that the *application* itself required almost no modifications. In addition, we were able to change the method of compression used by simply modifying the application to "plug in" the desired compression algorithm. These features provide easy access to application developers who do not want to spend their time learning the details of the media devices or who want to dynamically change their applications at run time.

2. DAVE MODEL

An illustration of a video conferencing application based on the DAVE model is shown in Figure 1. The major components of the DAVE model include an application programming interface (API), a connection manager, an object manager, and device objects. Application developers are responsible for making DAVE API calls to create and connect the device components necessary for their applications and have the option of adding new devices to the system. The object manager (OM) handles all real-time activities, manages device objects, and interfaces to the connection manager. The connection manager (CM) is responsible for all non-real-time activities, such as resource allocation and authorization. Details of these components and the overall model are discussed in the following sections.

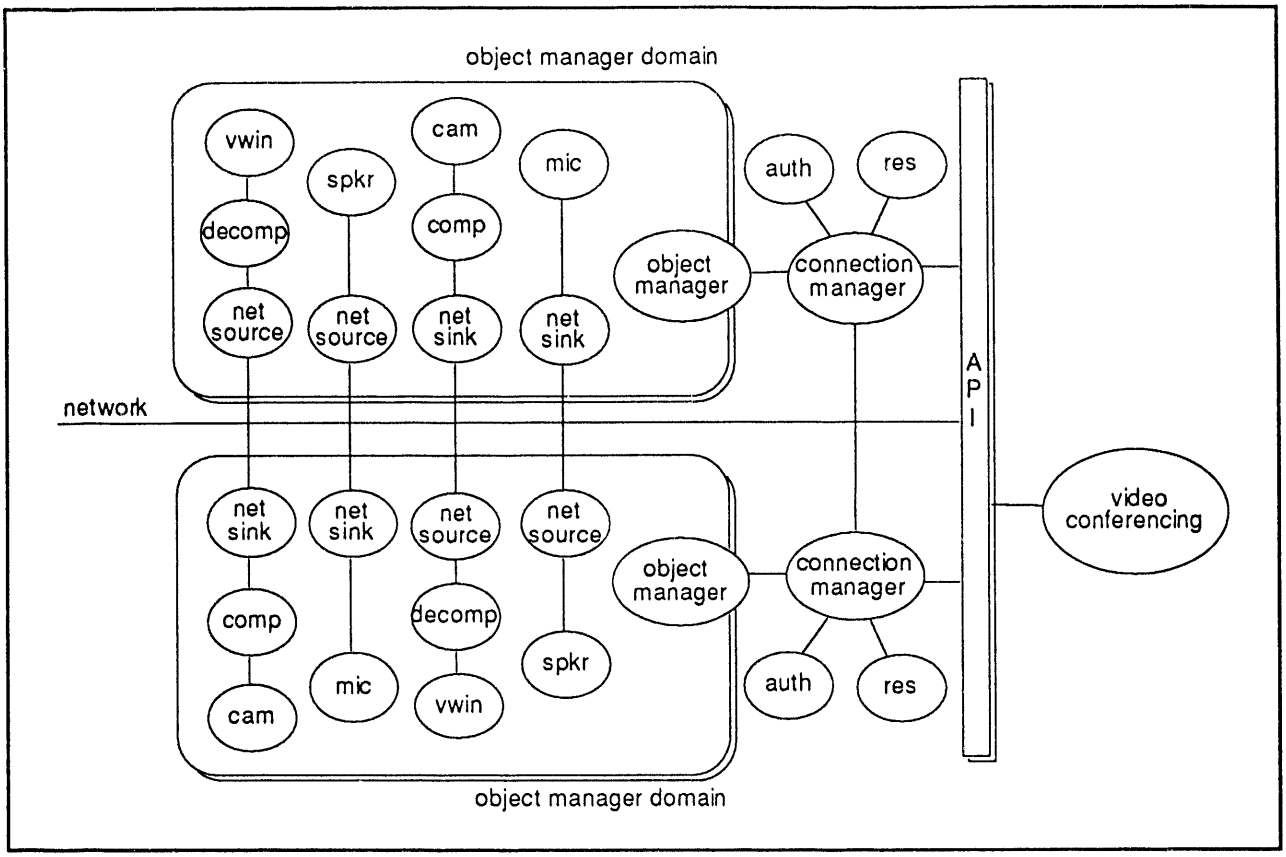


Figure 1. Video conferencing application based on the DAVE model.

2.1 PROGRAMMING MODEL

The usefulness of a well-defined API was shown by Bellcore with their Touring Machine[2]. DAVE provides a simple and intuitive programming paradigm for application developers to use. Since multimedia applications typically interact with a variety of physical devices, such as VCR's and remote controls, DAVE emulates that physical environment as much as possible through the use of software abstractions. The API is intended not only to provide a simple programming interface, but also to facilitate reusability of software. As underlying portions of the model change, a high degree of portability between media spaces in DAVE will continue to exist.

DAVE defines a plug-and-play programming paradigm. This paradigm was designed to emulate the analog audio-video environment with which most developers are familiar. Using this interface, programmers create distributed devices. These devices are software abstractions representing real physical devices, and are created on host machines either locally or remotely over a network through a Create call to the API. The programmer must specify the host machine and the device type. A unique object ID is returned. These distributed objects can then be connected by an API call to connect devices where the ID of the source and sink objects are specified. After the devices are connected, the system is told to enter the run state, and the devices are periodically sampled. The notion that devices are essentially objects that get sampled at a user-specified sampling interval is central to the DAVE model. Other API calls include the ability to send a command message to any device using its object ID as a reference, to allocate resources, and to query the system for information concerning available resources.

The video conferencing application shown in Figure 1 shows several devices connected together to form an application network that spans the physical network. In this case, the network connects two machines. On each machine, a camera is plugged into a compression device, which is plugged into a network sink device. A network source device on the other machine reads the media data from the network and passes the data to a decompression device. The decompression device is connected to a video window. By making these connections, we have put together a video stream.

This API is also designed to allow for dynamic applications. For example, if a new video compression algorithm is developed, it can easily be plugged into the media stream Using this programming model, determining which compression algorithm to plug in can be made at run time. In addition, a local frame capture application to store video to disk could be changed to a application that provides video to other machines on the network by replacing the Sink device. In the first case, the local video stream would begin with a Camera device and end with a file system Sink device. To change this to a video broadcast application, the video stream would begin with the Camera device but the file system Sink device would be replaced by a network Sink device. This provides for a high level of object reusability as well as flexibility because the same device objects can be plugged into many applications.

2.2 APPLICATION PROGRAMMING INTERFACE (API)

The API is a key component in the DAVE model. It is through this programming interface that application developers are able to create complex distributed applications using only a few application calls. The API provides an interface between the connection manager and the application. It uses lexical analysis to identify valid input and then checks for valid sequencing of the input from the API call by parsing the input. The API then puts the information into the correct format and passes it to the connection manager. Information received from the connection manager is returned through the API. A typical call is to create a device object. This request gets passed to the connection manager and then to the object manager. The object manager returns either an object ID, which is used as a device handle, or an error condition. Some of the basic API calls used in DAVE are listed in Table 1.

Call	Parameters
Create	ObjectType, ...
Delete	ObjectType, ...
ConnectObject	SourceObject, SinkObject, ...
DisconnectObject	SourceObject, SinkObject, ...
Connect	MachineID, ...
Allocate	ObjectType, ...
ResourceQuery	ResourceType, ...
Init	
Command	ObjectID, ...

Table 1. DAVE API calls.

An abbreviated code sample for the development of the audio portion of the application described in Figure 1 is listed in Figure 2.

From this code example, it can be seen that DAVE provides a very simple programming interface for application developers. This "plug-and-play" programming paradigm is one of the unique features and strengths of DAVE.

```
SessionIDA = Init( HostA, AccessInfo );
SessionIDB = Init( HostB, AccessInfo );
Status = Connect( HostA );
Status = Connect( HostB );
MikeA = Create( HostA, TYPE_MIKE );
MikeB = Create( HostB, TYPE_MIKE );
SpeakerA = Create( HostA, TYPE_SPEAKER );
SpeakerB = Create( HostB, TYPE_SPEAKER );
NetSinkA = Create( HostA, TYPE_NETSINK, TYPE_HOST, HostB );
NetSinkB = Create( HostB, TYPE_NETSINK, TYPE_HOST, HostA );
NetSourceA = Create( HostA, TYPE_NETSINK, TYPE_HOST, HostB );
NetSourceB = Create( HostB, TYPE_NETSINK, TYPE_HOST, HostA );
Status = ConnectObject( MikeA, NetSinkA );
Status = ConnectObject( MikeB, NetSinkB );
Status = ConnectObject( SpeakerA, NetSourceA );
Status = ConnectObject( Speaker, NetSourceB );
Status = Run( HostA );
Status = Run( HostB );
```

Figure 2. Sample Code Segment

2.3 CONNECTION MODEL

Within the DAVE model, there exists the concept of a session. A session is an association that exists between an application and a connection manager. A connection manager can participate in exactly one DAVE session. A session is created by the application through an API call to initialize the session. At this point, the connection manager and object manager are queried for their state. If they are not in a Connected state, then a session ID is created by the connection manager and returned to the application. At this point, the connection manager enters an Initializing state. Access control information is provided along with the API call to initialize a session. This information is checked before any connections can be established.

This simple model allows great flexibility for developing applications that require different control configurations. Applications that are centrally controlled are preferred in some environments[3]. This centralized model is better suited for applications that require centralized functions for accounting or security purposes, or distributed computing applications where a single source of information is being processed in parallel by many external slaves[4]. Other applications, such as multi-party video conferencing, are better modeled using a peer-to-peer model where no single person is controlling the overall conference[5]. The current model supports either of these two general configurations. Future work will focus on extending this model by providing for the merging of sessions and for the coexistence of multiple sessions on a single host. Currently, a DAVE host can only be involved in a single DAVE session.

2.4 CONNECTION MANAGER

The connection manager is responsible for all functions that are not real-time. These functions include resource allocation, local security checking, exception handling, and state information, including configuration management. The connection manager exists as a UNIX daemon called the Connection Manager Daemon (CMD). It acts as an interface between the application and the Object Manager Daemon discussed above. All requests pass through the CMD, which allows it to validate, store, forward, and execute those requests. In addition, the connection manager must negotiate and ensure the correct sequence of requests required to build the application system. In general, the CMD serves as a clearinghouse for non-real-time communications for its host machine. The CMD is currently implemented with only the basic functionality to allow for the creation and execution of sessions and applications. Future implementations of the CMD will have enhanced functionality to support resource allocation and security.

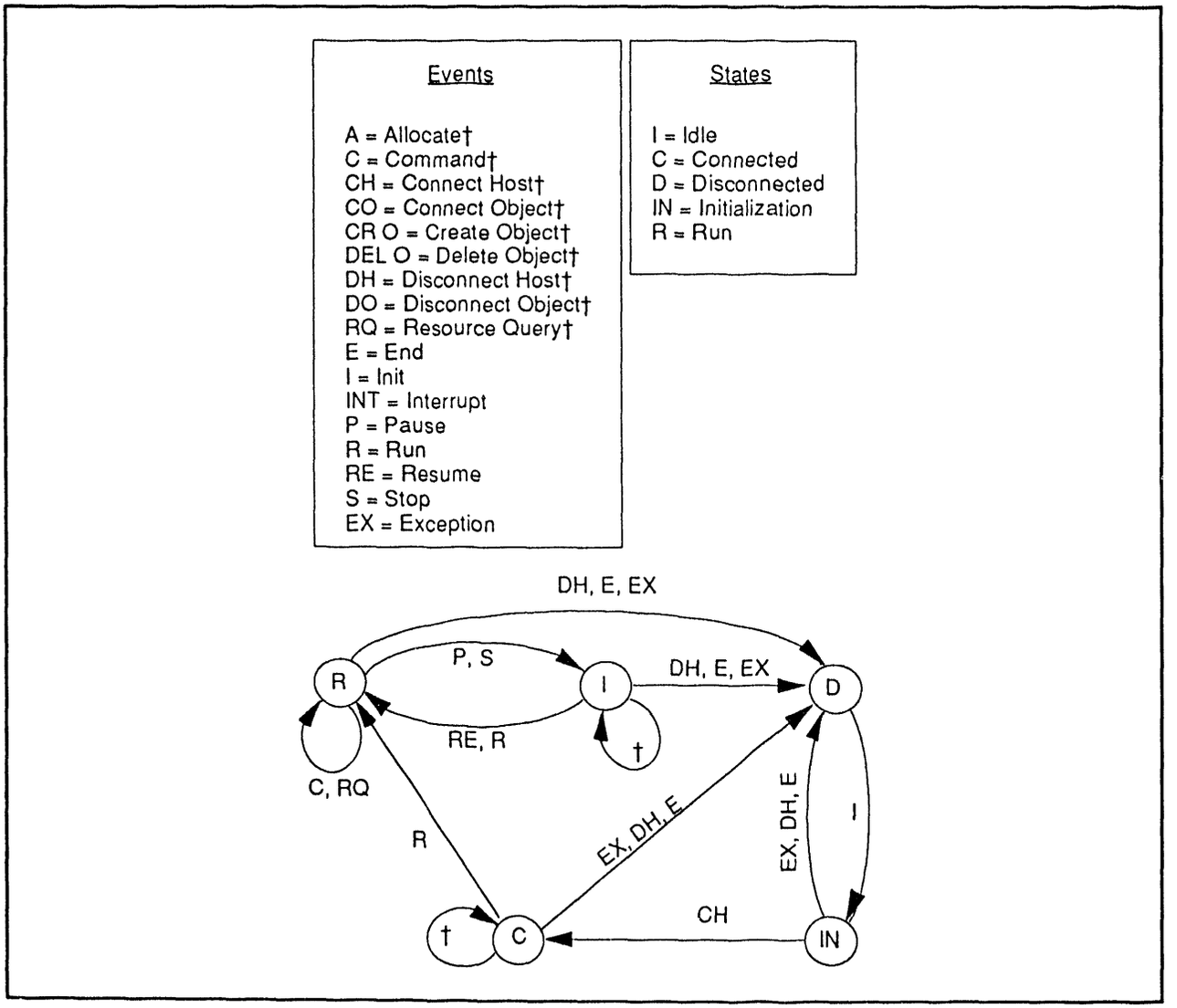


Figure 3. State diagram for Connection Manager

2.5 EXECUTION MODEL

DAVE was developed to emulate the physical media environment that it abstracts. This is seen in the plug-and-play programming paradigm and in the execution model for DAVE. The execution model used by DAVE can be classified as data-driven, rate-activated, node-limited and data-independent. Each of these classifications is explained below.

DAVE is designed to be driven by device data, or "data-driven." This means that the presence of data at a device causes the device to be processed. The usefulness of data-driven models for building applications has been demonstrated by numerous scientific visualization and image processing applications[6]. As data becomes available at the devices, it is passed through the system until its usefulness ends.

This processing of the data has beginning and end points. These points are called true sources and true sinks, respectively. A true source is a device that creates data and can be identified in the model by the fact that it has no other device as its source; a typical example is a camera device. A true sink is any device that has no other device as its sink; a typical example is a network sink device. A true source and its corresponding sink must reside on the same host.

However, true sources are not data-driven, but "rate-activated," which means that they are sampled at a rate specified by the application. For example, a Camera might have a sample rate of 15 samples per second. The object manager in DAVE attempts to honor that rate to the extent that it is possible. There are realistic constraints to this technique. For example, sampling rate cannot exceed the computer clock frequency. In addition, there are usually other activities taking place on a computer that compete for operating system resource allocation and scheduling. DAVE does not require special operating system support for real-time applications, but it would benefit from it[7].

The execution model for DAVE is "node-limited," which means that for any given true source of data, the true sink for that data stream is guaranteed to exist on the same host. There is some state information that can be carried to outside hosts, and typically is, but that is independent of the DAVE execution model. Nodal limitation turns out to be an advantage in terms of the types of distributed applications that can be developed, because of increased flexibility. Developers can create distributed applications that are either peer-to-peer or centrally organized. One application that takes advantage of a peer-to-peer organization is a multicast video application. Here the transmitter and receivers have no connection to each other; They each run as independent applications. At the other extreme is remote experiment control. Here the creation and control of media devices on several machines would be performed from a single central application.

DAVE is "data-independent" because sampling is done by the object manager independent of the type of data being processed by the device being sampled. True sources are responsible for providing message space, and each device in the chain has the option of either using the buffer provided by its source or providing its own. Type checking is not performed in the system because devices are connected independent of the type of data they are processing. Thus, plugging a camera device into a speaker device is allowed, and it is up to the application programmer to deal with the results. The ability to separate media type from the execution model provides for the creation and addition of additional media types. This media independence makes DAVE unique from other distributed multimedia system.

Once the sampling of devices has begun, the user application is free to do other processing. The application has the option of registering a callback routine to handle exception conditions returned from DAVE. These events could come from a device or one of the managers. Exception conditions include an object ID that will signal the source of the error, and they may also include other device-specific information.

DAVE supports one other type of execution that is an exception to the model described above. It allows for the creation of separate processes that run independently of the rest of the DAVE system. Control over these processes is limited to creation and deletion. This capability is provided to give developers the ability to exec separate UNIX processes and later to kill them.

2.6 OBJECT MANAGER DAEMON

An object manager is responsible for managing the distributed objects or devices. Management, in this case, means instantiation, deletion, sampling, and configuration. All real-time functions are handled by the object manager. The manager gets requests passed to it through the CMD, and it acts upon those requests. In addition, it passes back to the application exception conditions to the application through the connection manager and the API. The object manager exists as a UNIX daemon, called the Object Manager Daemon (OMD), and receives requests by way of sockets.

The choice of implementing the OMD and DAVE devices as a single process, as opposed to a separate process for each device, was chosen for efficiency and ease of implementation. By using a single process we were able to reduce the delays associated with process context switching, and reduce the number of memory copies required. For a typical application there are no memory copies performed by the OMD or the devices. In the case of network devices the operating system may or not perform some memory copies, but these are out of our control.

The OMD keeps track of application objects and device connections. As device Create and Connect requests are received, it instantiates the specified devices and keeps track of the logical device connections made through the plug-and-play API. This information is stored in an associative array. The array keeps track of which devices are connected and which are true sources (devices with no sources of their own) and true sinks (devices with no sinks of their own).

SOURCE DEVICES	SINK DEVICES		
	microphone	*filter*	*network*
microphone	0	1	0
filter	0	0	1
network	0	0	0

Table 2. Logical device connections for audio broadcast

Table 2 is an example of an array for a simple audio broadcast application. There are three devices: a microphone, an audio filter, and a network sink. When in the Run state, the OMD scans this array for true sources. In this case, the microphone is not a sink for any other device so it is a true source. It then checks the device to see if it is time to sample it. If the device is ready, the OMD samples it and passes the return information to the sink device. The OMD continues this process until a true sink is reached. In this example, the true sink is the network sink. Any device can have multiple sources, but only one sink. After sampling the device, the sample time is updated. In addition, the sample time of the Message buffer is updated. This information can be used for synchronization.

The OMD enters the Run state via a Run API call. The OMD cannot enter a Run state until it first enters the Connect state. The OMD can only be connected to a single CMD at any given time. Once the Run state is entered, sampling of the objects using the execution model described above begins. This sampling continues until an exception condition occurs that causes the Run state to be

exited, or until a Pause, Disconnect, or Stop command is received. Exception conditions recognized by the OMD are reported back through the connection manger The OMD also provides miscellaneous housekeeping chores such as calling the device Init routine at instantiation, passing messages between devices, and maintaining a connection to a CMD.

2.7 DEVICE CLASS HIERARCHY

DAVE was designed to provide device extensibility. To enable this, the model provides a well-defined interface to device objects, along with a device class hierarchy. At the top of this hierarchy is a base Dev class. This class contains the attributes and methods that are required by the object manager to manage these devices. These attributes include a device ID, a sampling rate, and time stamps that indicate the last and next sample time. In addition to the attributes defined, there are five virtual functions that provide the interface between the object manager and the device itself. These functions include a command function and a process function. The command function (cmd) provides a means to send a Command message type to the device, and for a device to return exception information to the application. A subset of the device class hierarchy as it exists in DAVE is shown in Figure 4.

Developers that want to create additional devices can derive them from the Dev base class. This inheritance gives the OMD the information and control that it needs to manage the device. Each device must define the virtual functions defined in the base Dev class. The process function (proc) is called at sample time for the device. If the device does not have a sample time specified for it, the OMD uses a default value. The initialize function (init) is called by the OMD when it creates a device, and a Command message is optionally passed to it.

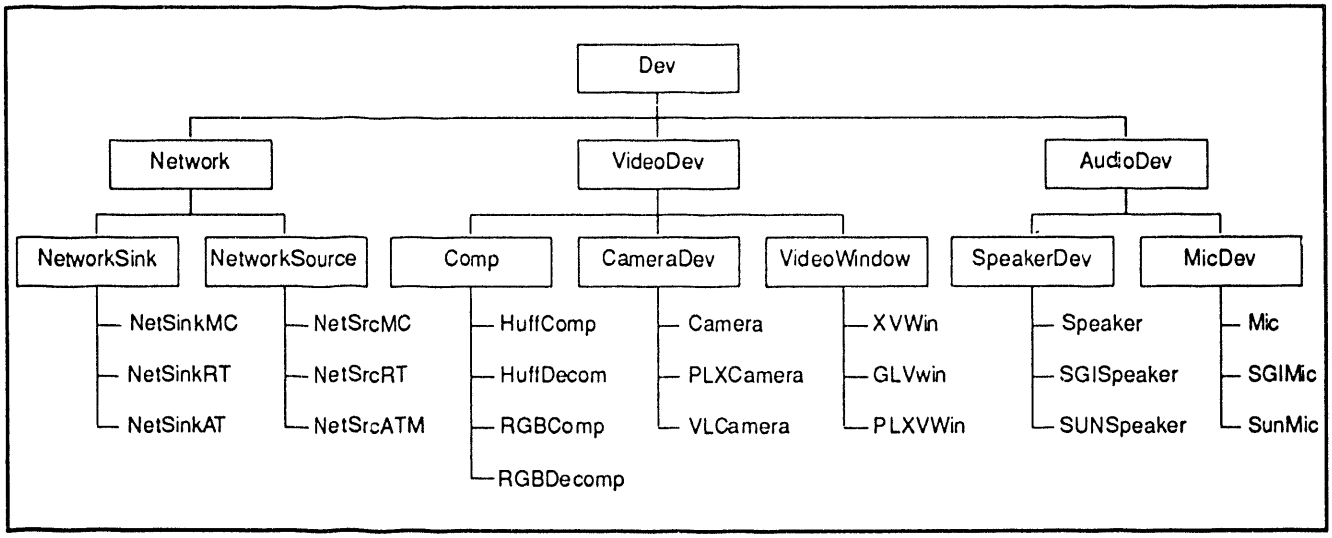


Figure 4. Device class hierarchy.

2.8 MESSAGE HIERARCHY

In addition to the device class hierarchy in DAVE, there is also a message hierarchy. This message hierarchy is used to provide media type extensibility. Application developers can define their own media data types using this hierarchy. DAVE currently provides data types that include audio and video message classes. Other media types can be defined by inheriting from the Message base class. All of the attributes needed by the object manager to manage these buffers are defined in the base class. In addition, dynamic binding is used to provide a function interface to the data storage

area in the message. A partial listing of the Message class hierarchy as it currently exists is shown in Figure 5.

Messages are the means for passing data inside the DAVE model. All DAVE devices accept and return a pointer to a Message from their Process function. In addition, the Command Message class provides a general mechanism for passing commands to and from devices in the model. No other data types can be manipulated by the OMD or by DAVE devices. To facilitate the management of messages, there is a MsgMgr object provided with DAVE.

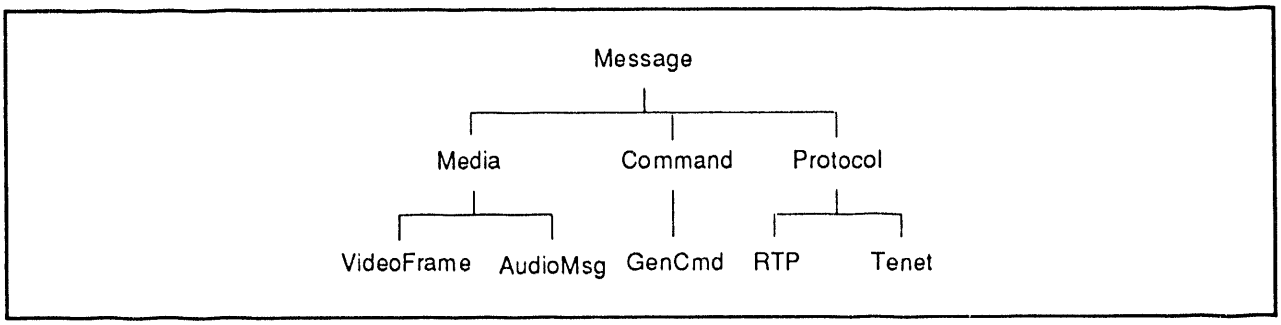


Figure 5. Message class hierarchy.

3. DISCUSSION OF THE MODEL

The DAVE model is unique in that it provides the combination of a distributed plug-and-play API, offers device and media extensibility, is object oriented, uses traditional UNIX network facilities for transmission, and uses existing workstation audio and video hardware that is commonly available on many workstations. We have seen many individual desktop multimedia applications emerging but these have been monolithic. These applications, like vat[8] and nv[9], have proven usefulness of collaborative tools but do not provide the flexibility and reuseability needed to treat multimedia capabilities as network resources. Computer vendors, such as SGI and SUN, have developed products that are collaborative tools similar to vat and nv, but these also are static applications. Again, they fail to treat multimedia capabilities as shared network resources and do not provide a distributed development environment.

The strengths of the DAVE model come from the programming interface, the reusability of objects, media and device extensibility, and network independence. During the development of our applications, we found the actual code development was simple and elegant by using the plug-and-play API. Many of the objects were reused several times without modification. The ability to provide network independence by viewing the network as just sinks and sources allowed us to run our applications over traditional networks such as Ethernet and FDDI, as well as future networks such as Asynchronous Transfer Mode (ATM) using direct AAL5 calls. DAVE is also very good at hiding the details of individual devices from the application developer. As an example, developers can use Microphone and Speaker objects without having to worry about what to do about silence detection and other characteristics specific to a device. Finally, because the OMD and devices run inside a single process we were able to keep memory copies to a minimum (usually none) and minimize scheduling delays caused by the operating system. Measurements of application performance characteristics are under way.

Bellcore's Touring Machine[2] was one of the first research efforts to realize the advantage of an API to provide support for the development of a wide range of applications. The Touring Machine was designed around analog transmission and switching. DAVE uses standard UNIX networking

for media transmission. The DAVE approach does not require the purchase of special cable plant or exotic hardware and supports both multicast and broadcast of data thereby increasing interoperability and portability. AT&T's Rapport[1] appears to somewhat address the issue of media independence through software abstractions. Rapport, like the Touring Machine, is based on analog transmission.

More directly related to DAVE is the IMA's Multimedia System Services specification (MSS). Similar to DAVE, the MSS integrates multimedia resources into a distributed environment. The MSS also provides a "plug-and-play" programming environment for application developers. There are several other areas were the two systems differ.

DAVE has a well-defined execution model as well as object model. The MSS defines only an object model; DAVE does not require the use of an Object Request Broker. We did not want to rely an ORB's existence for DAVE; the MSS requires an ORB. DAVE was designed to be data independent allowing for easy extensibility, whereas the MSS defines data types to allow for type checking. DAVE defines a simple standard system/device interface to hide device specifics, whereas the MSS provides a flexible developer defined device specific interface.

4. DAVE APPLICATIONS AND STATUS

In order to provide proof-of-concept of the DAVE model, we developed two separate applications that use all features of the model. The first application is video conferencing, and the second is one-to-many audio/video. Both applications are capable of controlling all DAVE objects from instantiation through modification to deletion. For example, both applications can instantiate a speaker object, change its output volume, change its output sampling rate, switch between headphone and speaker jacks, and destroy the object when completed. Using the DAVE application programming interface, a virtual connection can be established between a microphone on one machine, through the necessary network objects, to a speaker on another machine.

The broadcast video application is capable of "tuning in" to multiple channels, similar to a home television tuner. When the user selects a "channel change," new connections are established to the appropriate audio and video network objects, and the user sees a new channel. The workstation being used to view the broadcast video does not need to be video-capable, i.e., have a video capture board, to receive the broadcast.

The user interface for the video conferencing application is used to establish a connection to a user on a remote video-capable workstation. The user specifies the remote machine and username with whom the user wishes to have a conference, and, if the remote user accepts the call, a connection is built. At this point, the user is capable of adjusting the local input and output volume controls, the sampling rate, and all video quality parameters such as size, frame rate, and compression quality.

Both of these applications required a minimal amount of coding. Excluding the user interface components, the video conferencing and audio-video broadcast applications required between 100-200 lines of code. This demonstrates the power of the DAVE API. In addition, both applications used the same device objects. No additional devices were required to be written. This points to a very high level of code reuseability. A video tape of these applications is available and can be obtained by emailing any of the authors.

The DAVE software is still under development. Improvements in functionality and fixes are being performed for the CMD and the OMD. Much of our work is currently focused on improving the API and the CMD in the areas of exception callbacks and resource specifications.

5. CONCLUSIONS

DAVE has proven to be a useful model for developing distributed multimedia applications. This has been shown by developing desktop video conferencing and other distributed applications based on DAVE. These applications take advantage of the simple plug-and-play programming paradigm. The applications were simple to design and develop using DAVE. Each application uses the same distributed DAVE objects, which provides a very high level of reusability for these objects.

The strengths of the DAVE model come from the programming interface and the ability to provide a high level of abstraction for devices through object oriented techniques. The programming paradigm allows application developers to treat media devices, like cameras and microphones, as distributed resources akin to the way graphics and windows are used today on workstation. This flexibility and level of accessibility allows developers to easily integrate multimedia into our existing distributed environment. The ability to "plug-and-play" multimedia resources through the use of a few API calls greatly simplified the development of applications and at the same time hide device specific interfaces from application developers.

REFERENCES

[1] S. Ahuja, J. R. Ensor, S. E. Lucco. "A Comparison Of Application Sharing Mechanisms in Real-time Desktop Video Conferencing". Proceedings IEEE Conference on Office Information Systems. 1990.

[2] Gita Gopel, Gary Herman, and Mario P. Vecchi. "The Touring Machine: Toward A Public Network Platform For Multimedia Applications". Bellcore. Morristown, NJ.

[3] XShell 2.2 Release Notes. Xpersoft Corporation, San Diego, CA. 1992.

[4] J. F. Macfarlane, R. C. Armstrong, R. E. Cline, Jr., and M. L. Koszykowski. "Application of Parallel Object-Oriented Environment and Toolkit (POET) to Combustion Problems." Internal Report. Sandia National Laboratories, Livermore, CA. 1992.

[5] Eve M. Schooler. "A Distributed Architecture for Multimedia Conference Control". ISI Research Report ISI/RR-91-289. 1991.

[6] The Khoros Group, "The Khoros Users Manual". Department of Electrical and Computer Engineering, University of New Mexico, Albuquerque, NM. 1987.

[7] S. M. Stevens. Proceedings "Next Generation Network and Operating System Requirements for Continuous Time Media". Second International Workshop on Network and Operating System Support for Digital Audio and Video. 1991.

[8] Van Jacobson. VAT - Visual Audio Tool manual page. Lawrence Berkeley Laboratories, Berkeley, CA.

[9] Ron Fredericks. nv - Network Video manual page. Xerox PARC. Palo Alto, CA. 1992.

UNLIMITED RELEASE

INITIAL DISTRIBUTION:

9003	D. L. Crawford, 8900
9011	K. E. Palmer, 8901
9040	G. Gutierrez, 8902
0803	W. D. Swartz, 8903
0805	A. R. Iacoletti, 8904
9011	P. W. Dean,8910
9011	J. C. Berry, 8910
9012	D. H. Ching, 8910-1
9011	R. E. Cline, 8920
9011	J. A. Friesen, 8920
9011	E. W. Knightly, 8920
9011	R. F. Mines, 8920 (10)
9011	C. L. Yang, 8920
9001	J. C. Crawford, 8000
	Attn: E. E. Ives, 5200
	J. B. Wright, 5300
	M. E. John, 8100
	R. J. Detry, 8200
	W. J. McLean, 8300
	L. A. Hiles, 8400
	P. N. Smith, 8500
	L. A. West, 8600
	R. C. Wayne, 8700
	T. M. Dyer, 8800
9022	Mail Distribution (8533-1) for OSTI (10)
9022	Mail Distribution (8533-1)/Technical Library Processes, MS-0899, (7141) (3)
9018	Central Technical Files (3)

DATE
FILMED
9/30/94

END